本节内容

栈

栈

浏览器的后退和前进键

水桶里的水倒入时后倒进去的水肯定先倒出

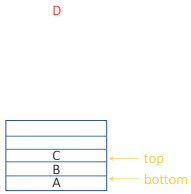
弹夹装弹

栈

栈（Stack）：只允许在一端进行插入或删除操作的线性表。

栈顶（Top）：线性表允许进行插入和删除的那一端。

栈底（Bottom）：固定的，不允许进行插入和删除的另一端。



Tips：
1.栈是受限的线性表，所以自然具有线性关系。
2.栈中元素后进去的必然先出来，即后进先出
LIFO（Last In First Out）

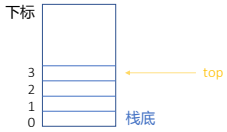
顺序栈

栈是线性表的特例，那栈的顺序存储也是线性表顺序存储的简化。栈的顺序存储结构也叫作顺序栈。

回忆一下：之前在实现顺序表时，我们用的是数组来实现。那实现顺序栈是不是可以也可以用数组呢？

由于栈是受限的线性表，所以顺序栈需要在顺序表的基础上做点修改。

想一下：数组哪一端做栈底比较好？



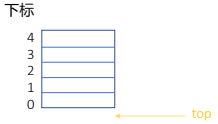

```
#define MaxSize 50        //定义栈中元素的最大个数
typedef struct{
        Elemtype data[MaxSize];      //存放栈中元素
        int top;                     //栈顶指针
} SqStack;                           //顺序栈的简写
```

1. Top值不能超过MaxSize
2. 空栈的判定条件通常定为top==-1，满栈的判定条件通常为top==MaxSize-1，栈中数据元素个数为top+1

顺序栈的操作

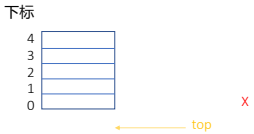
1.判空：

```
bool StackEmpty(SqStack S){
        if(s.top==-1)  return  true;
        else              return  false;
}
```



2.进栈：

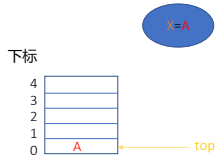
```
bool Push(SqStack &S ，ElemType x){
        if(S.top==MaxSize-1) return  false;
        S.data[++S.top]=x;    return  true;
}
```



顺序栈的操作

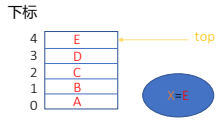
3.出栈：

```
bool Pop(SqStack &S ，ElemType &x){
        if(S.top==-1)  return  false;
        x=S.data[S.top--];
      return  true;
}
```



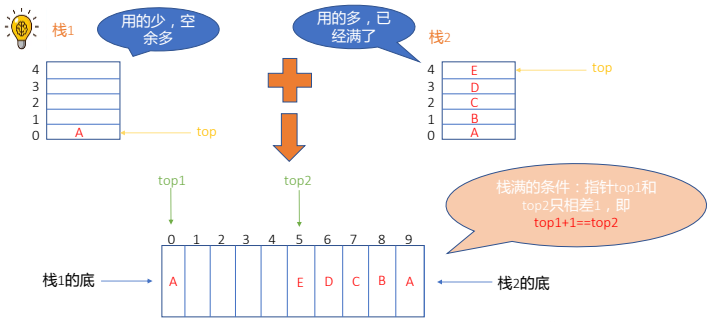
4.读取栈顶元素：

```
bool GetTop(SqStack S ，ElemType &x){
        if(S.top==-1)        return  false;
        x=S.data[S.top];
      return  true;
}
```



共享栈

顺序栈的存储空间大小需要事先开辟好，很多时候对每个栈各自单独开辟存储空间的利用率不如将各个栈的存储空间共享



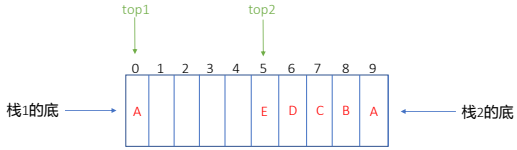
共享栈

共享栈的结构：

```
#define MaxSize 100      //定义栈中元素的最大个数
typedef struct{
        Elemtype data[MaxSize];      //存放栈中元素
        int top1;                    //栈1栈顶指针
        int top2;                    //栈2栈顶指针
} SqDoubleStack;                     //顺序共享栈的简写
```

共享栈的操作：（进栈）

```
bool Push(SqDoubleStack &S ，ElemType x，int stackNum){
        if(S.top1+1==s.top2) return  false;   //栈满
      if(stackNum==1)   S.data[++S.top1]=x;   //栈1有元素进栈
        else if(stackNum==2)   S.data[--S.top2]=x;  //栈2有元素进栈
      return  true;
}
```



链式栈

栈是线性表的特例，线性表的存储结构还有链式存储结构，所以也可以用链表的方式来实现栈。栈的链式存储结构也叫作链栈。

回忆一下：每个单链表都有头指针，对于栈来说，栈顶指针也是必须的。

可以将头指针当做栈顶指针，所以栈顶放在单链表的头部。


```
typedef struct SNode{
        Elemtype data;          //存放栈中元素
        struct SNode *next ;  //栈顶指针
} SNode , *SLink                //链栈的结点
typedef struct LinkStack{
        SLink top;              //栈顶指针
        int count;              //链栈结点数
}LinkStack                      //链栈
```

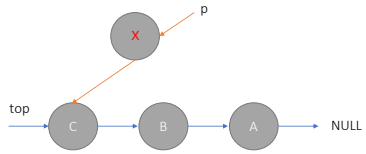
1.链栈一般不存在栈满的情况。

2.空栈的判定条件通常定为top==NULL；

链式栈的操作

1.进栈

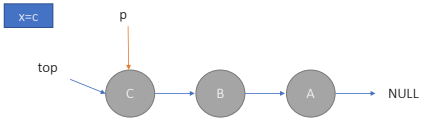
```
bool Push(LinkStack *S, ElemType x){
        SLink p=(SLink)malloc(sizeof(SNode));     //给新元素分配空间
        p->data=x;                                //新元素的值
        p->next=S->top;                           //p的后继指向栈顶元素
        S->top=p;                                 //栈顶指针指向新的元素
        S->count++;                               //栈中元素个数加1
        return true;
}
```



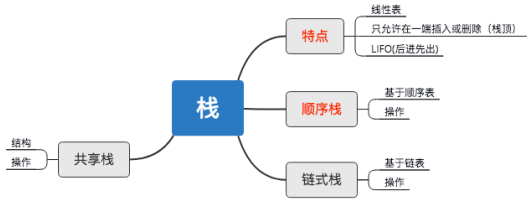
链式栈的操作

2.出栈

```
bool Pop(LinkStack *S, ElemType&x){
        if(S->top==NULL) return false;
        x=S->top->data;                   //栈顶元素值
        Slink p=S->top;                   //辅助指针
        S->top=S->top->next;              //栈顶指针后移
        free(p);                          //释放被删除数据的存储空间
        S->count--;                       //栈中元素个数减一
        return true;
}
```



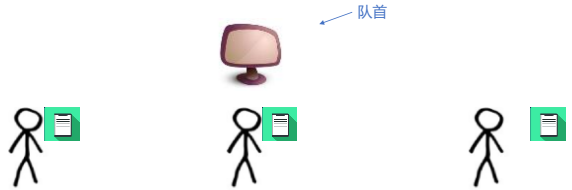
总结



本节内容

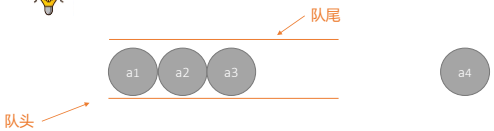
队列

队列



队列

队列是只允许在一端进行插入，而在另一端进行删除的线性表



队头（Front）：允许删除的一端，又称为队首。
队尾（Rear）：允许插入的一端。

先进入队列的元素必然先离开队列，
即先进先出（First In First Out）简称FIFO

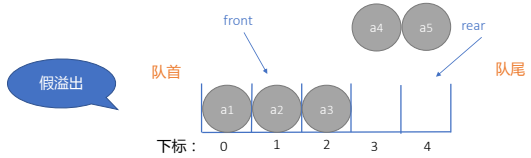
栈中元素后进去的必然先出来，即后进先出LIFO（Last In First Out）

顺序队列

用数组来实现队列，可以将队首放在数组下标为0的位置。

不要求从数组首位开始存储队列。也就是说队首不一定要在数组下标为0的位置。那如何表示队首呢？

```
#define MaxSize 50                    //定义队列中元素的最大个数
typedef struct{
    ElemType data[MaxSize];           //存放队列元素
    int front , rear;                 //队头指针和队尾指针
} SqQueue;
```



循环队列

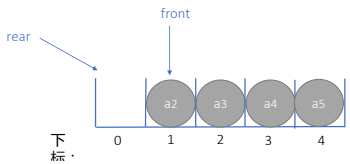
把数组“掰弯”，形成一个环。Rear指针到了下标为4的位置还能继续指回到下标为0的地方。这样首尾相连的顺序存储的队列就叫循环队列

入队：rear=(rear+1)%MaxSize

出队：front=(front+1)%MaxSize

此时出现了新的问题，rear和front指针值相同，我们说过空队列时front等于rear，现在队列满了也是front等于rear，那如何分辨队列是空还是满呢？

方法一：设置标志位flag，当flag=0且rear等于front时为队列空，当flag=1且rear等于front时为队列满。



循环队列

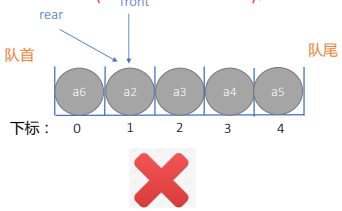
方法二：我们把front=rear仅作为队空的判定条件。当队列满的时候，令数组中仍然保留一个空余单元。我们认为这种情况就是队列满了。

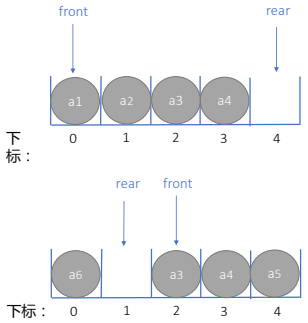
那么栈满时，有什么等量关系？

(rear+1)%Maxsize==front

队列中元素个数：

(rear-front+MaxSize)% MaxSize





循环队列的操作

1.入队：

```
bool EnQueue(SqQueue &Q,ElemType x){
        if((Q.rear+1)%MaxSize==Q.front)  return false;     //队满
        Q.data[Q.rear]=x;
        Q.rear=(Q.rear+1)%MaxSize;                                  return
true;
}
```

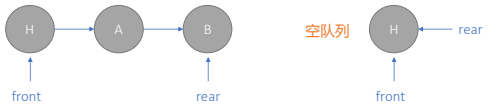
2.出队：

```
bool DeQueue(SqQueue &Q,ElemType &x){
        if(Q.rear==Q.front)  return false;                  //队空，报错
        x=Q.data[Q.front];
        Q.front=(Q.front+1)%MaxSize;
        return  true;
}
```


链式队列

队列的链式存储结构，其实就是线性表的单链表，只不过需要加点限制，只能表尾插入元素，表头删除元素。

为了方便操作，我们分别设置队头指针和队尾指针，队头指针指向头结点，队尾指针指向尾结点。

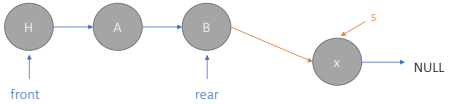



```
typedef struct{                  //链式队列结点
        ElemType data;
        struct LinkNode *next;
}LinkNode;

typedef struct{                  //链式队列
        LinkNode *front,*rear;   //队头和队尾指针
}LinkQueue;
```

链式队列的操作

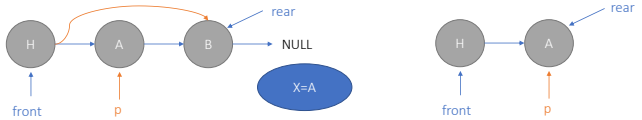
1.入队：我们知道队列只能从队尾插入元素，队头删除元素。于是入队就是在队尾指针进行插入结点操作。链队的插入操作和单链表的插入操作是一致的。




```
void EnQueue(LinkQueue &Q，ElemType x){
   s=(LinkNode *)malloc(sizeof(LinkNode));
   s->data=x;
   s->next=NULL;
   Q.rear->next=s;
   Q.rear=s;
}
```


链式队列的操作

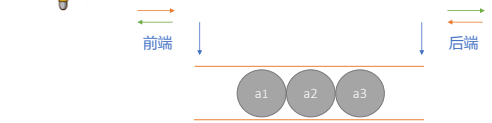
2.出队：出队就是头结点的后继结点出队，然后将头结点的后继改为它后面的结点。




```
bool DeQueue(LinkQueue &Q,ElemType &x){
   if(Q.front==Q.rear) return  false;    //空队
   p=Q.front->next;
   x=p->data;
   Q.front->next=p->next;
   if(Q.rear==p)  Q.rear=Q.front;                    //若原队列中只有一个结点，删除后变空
   free(p);
   return  true;
}
```


双端队列

双端队列：双端队列是指允许两端都可以进行入队和出队操作的队列



输入受限的双端队列

输出受限的双端队列

本节内容

栈的应用

栈的应用

1、括号匹配：假设有两种括号，一种圆的()，一种方的[]，嵌套的顺序是任意的。

([ ]) [([ ][ ])] 合法

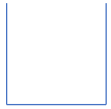
[(]) (()] 非法

我更急

((

[([ ] [ ])]

[([ ] [ ])]



栈的应用

```
bool Check(char *str ){
  stack s;
  InitStack(s);
  int len=strlen(str);  //字符串长度为len
  for(int i=0;i<len;++i)
  {
     char a=str[i];
     switch(a){
      case '(':
      case '[':
          Push(s,a);
          break;
     case ')':
         if( Pop(s)!='(' ) return false;  //出栈顶。如果不匹配直接返回不合法
         break;
     case ']':
          if( Pop(s)!='[' ) return false;
          break;
     }
  }
 if(Empty(s))  return true;  //匹配完所有括号最后要求栈中为空
 else return false
}
```

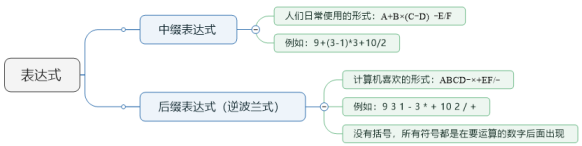
算法思想：若是左括号，入栈；若是右括号，出栈一个左括号判断是否与之匹配；检验到字符串尾，还要检查栈是否为空。只有栈空，整个字符串才是括号匹配的。

(( ]

栈的应用

2、表达式求值：

我们都知道四则表达式的计算法则：从左到右，先乘除后加减，有括号先算括号里的。那如何让计算机也懂这些运算法则呢？



后缀表达式为什么是计算机"喜欢"的形式呢？

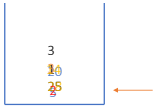
因为计算机可以通过栈来计算后缀表达式的值,从而计算表达式的值

栈的应用

规则：从左到右扫描表达式的每个数字和符号，遇到数字就进栈，遇到符号就将处于栈顶的两个数字出栈然后跟这个符号进行运算，最后将运算结果进栈，直到最终获得结果。

以计算(5+20+1*3)/14 为例 它的后缀表达式是5 20 + 1 3 * + 14 /

5 20 + 1 3 * + 14 /



1. 5+20=25
2. 1*3=3
3. 25+3=28
4. 28/14=2

所以最后的结果是2

栈的应用

那如何将中缀表达式转换成后缀表达式？

1.按运算符优先级对所有运算符和它的运算数加括号。(原本的括号不用加)
2.把运算符移到对应的括号后。
3.去掉括号。

((5 + 20 + 1 * 3) / 14)

(((5 20) + (1 3) *) + 14) /

5 20 + 1 3 * + 14 /

前缀表达式其实将运算符提到括号前面，其他都一样

栈的应用

3、递归：

要理解递归，你要先理解递归，直到你能理解递归。
如果在一个函数、过程或数据结构的定义中又应用了它自身，那么这个函数、过程或数据结构称为是递归定义的，简称递归。递归最重要的是递归式和递归边界。

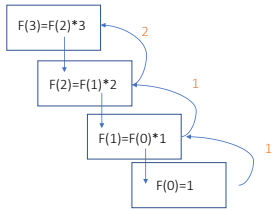
1）使用递归求解n的阶乘　n!=1*2*...*n

```
int F (int n) {
  if(n==0) return 1;
  else return n*F(n-1);
}
```

1.递归边界(n==0时 F(0)==1)

2.递归式　(F(n)=F(n-1)*n)

例如，求F(3)



栈的应用

2）求斐波那契数列的第n项

```
int  Fib (int n) {
   if(n==0)  return 0;
   else if(n==1)  return 1;
   else return Fib(n-1)+Fib(n-2);
}
```

$$fib(n)=\begin{cases} fib(n-1)+fib(n-2) & n>1 \\ 1 & n=1 \\ 0 & n=0 \end{cases}$$

1.递归边界 Fib(0)==0, Fib(1)==1

2.递归式　Fib(n)=Fib(n-1)+Fib(n-2)

例如，求F(4)